CITI Technical Report 00-1

Linux Kernel Hash Table Behavior: Analysis and Improvements

Chuck Lever, Sun-Netscape Alliance
<chuckl@netscape.com>

ABSTRACT

The Linux kernel employs hash table data structures to store high-usage data objects such as pages, buffers, inodes, and others. In this report we find significant performance boosts with careful analysis and tuning of four critical kernel data structures.

April 13, 2000

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via grants from the Sun-Netscape Alliance, Intel, Dell, and IBM. For more information, see our home page.

If you have comments or suggestions, email linux-scalability@citi.umich.edu.

Linux Kernel Hash Table Behavior:
Analysis and Improvements

Chuck Lever, Sun-Netscape Alliance
`<chuckl@netscape.com>`

1. Introduction

Hash tables are a venerable and well-understood data structure often used for high-performance applications because of their excellent average lookup time. Linux, an open-source POSIX-compliant operating system, relies on hash tables to manage pages, buffers, inodes, and other kernel-level data objects.

Why worry about the kernel's hash tables? As we show, Linux performance depends on the efficiency and scalability of these tables. On a small machine with 32M of physical RAM, a page cache hash table with 2048 buckets is probably large enough to hold all possible cache pages in chains shorter than three. However, a hash table this small cannot hold all possible pages on a larger machine with, say, 512M of physical RAM while maintaining short chains to keep lookup times quick. Keeping hash chains short is even more important on modern CPUs because of the effects of CPU cache pollution on overall system performance. Lookups on longer chains can push useful data out of CPU caches.

Hash tables depend on good average case behavior to perform well. Average case behavior relies on the actual input data more often than we like to admit, especially when using simple shift-add hash functions. Statistical examination of specific hash functions used in combination with specific real world data can expose opportunities for performance improvement.

It is also important to understand why hash tables are employed in preference to a more sophisticated data structure, such as a tree. Insertion into a hash table is **O(1)** if hashed objects are maintained in last-in first-out (LIFO) order in each bucket. A tree insertion or deletion is **O(*log* n)**. Object deletion and hash table lookup operations are often **O(n/m)** (where **n** is the number of objects in the table and **m** is the number of buckets), which approaches **O(1)** when the hash function spreads hashed objects evenly through the hash table and there are more hash buckets than objects to be stored. Finally, if designers are careful about hash table design, they can keep the average lookup time for both successful and unsuccessful lookups low (*i.e.* less than **O(*log* n)**) by using a large hash table and a hash function that thoroughly randomizes the key.

In this report we analyze several critical hash tables in the Linux kernel, and describe minor tuning changes that can improve Linux performance by a significant margin. The remainder of this report is organized as follows. Section 2 outlines our methodology. In Section 3, we separately examine four critical kernel hash tables and show how modifications to each hash table affect overall system performance. Section 4 reports results of combining the findings of Section 3. We discuss hash function theory as it applies on modern CPU architectures in Section 5, and Section 6 concludes the report.

2. Methodology

Our goal is to improve system throughput. Therefore, the final measure of performance improvement is benchmark throughput. However, there are a number of metrics we can use to determine the "goodness" of a hash function with a given set of real-life input keys. In this section, we describe our benchmark procedures and the additional metrics we use to determine hash function goodness.

We use the SPEC SDM benchmark suite to drive our tests [SPEC]. SDM emulates a multi-tasking software development workload with a fixed script of commands typically used by software developers, such as `cc`, `ed`, `nroff`, and `spell`. We model offered load by varying the number of concurrent instances of the script that run during the benchmark. The throughput values generated by the benchmark are in units of "scripts per hour." Each value is calculated by measuring the elapsed time for a given benchmark run, then dividing by the number of concurrent scripts running during the benchmark run. The elapsed time is measured in hundredths of a second.

We benchmark two hardware bases:

1. A four processor Dell PowerEdge 6300/450 with 512M of RAM and a single Seagate 18G LVD SCSI hard drive. This machine uses 450Mhz Xeon Pentium II's, each with 512K of L2 cache.
2. A two processor custom-built system with 128M of RAM and a pair of 2G Quantum Fireball hard drives. This machine uses 200Mhz Pentium Pro CPUs with a 256K external level 2 cache for each CPU, supported by the Intel i440FX chipset.

At the time of these tests, these machines were loaded with the Red Hat 5.2 distribution using a 2.2.5 Linux kernel built with egcs 1.1.1, and using glibc 2.0 as installed with Red Hat.

The dual Pentium Pro machine workloads vary from sixteen to sixty-four concurrent scripts. The sixteen script workload fits entirely in RAM and is CPU bound. The sixty-four script workload does not fit into RAM, thus it is bound by swap and file I/O. The four-way system ran up to 128 scripts before exhausting the system file descriptor limit because plain 2.2.5 kernels used in this report do not contain large `fdset` support. All of the 128 script benchmark fit easily into its 512M of physical memory, so this workload is designed to show how well the hash tables scale on large-memory systems when unconstrained by I/O and paging bottlenecks.

On our dual Pentium Pro, both disks are used for benchmark data and swap partitions. The swap partitions are of equal priority and size. The benchmark data is stored on file systems mounted with the "noatime" and "nosync" options for best performance. Likewise, on the four-way, the benchmark file system is mounted with the "noatime" and "nosync" options, and only one swap partition is used.

Hash performance depends directly on the laws of probability, so we are quite interested in the statistical behavior of the hash (*i.e.* its "goodness"). First, we generate hash table bucket size distribution histograms with special kernel instrumentation. This tells us:

- What portion of total table buckets are unused
- Whether a high percentage of hashed objects are contained in small buckets
- The worst-case (largest) bucket size
- Whether bucket size is normally distributed. A normal distribution indicates that the hash function spreads objects evenly among the hash buckets, allowing the hash table to approach its best average behavior

Second, we measure the average number of objects searched per bucket during lookup operations. We count the average number of successful lookups separately from the average number of unsuccessful lookups because an unsuccessful lookup requires on average twice as many key comparisons. These search averages are one of the best indications of average bucket size and a direct measure of hash performance. Lowering them means better average hash performance.

And finally, we are interested in how long it takes to compute the hash function. This value is estimated given a table of memory and CPU cycle times, estimating memory footprint and access rate, cache miss rate, and guessing at how well the instructions to compute the hash function will be scheduled by the CPU. We estimate these based on Hennessey and Patterson [Hen].

3. Four critical hash tables

In this section we investigate the response to our tuning efforts of four critical kernel hash tables. These tables included the buffer cache, page cache, dentry cache, and inode cache hash tables.

3.1 Page cache

The Linux page cache contains in-core file data while the data is in use by processes running on the system. It can also contain data that has no backing storage, such as data in anonymous maps. The page cache hash table in the plain 2.2.5 kernel comprises 2048 buckets, and uses the following hash function from include/linux/pagemap.h:

```
#define PAGE_HASH_BITS 11
#define
    PAGE_HASH_SIZE (1 << PAGE_HASH_BITS)

static inline unsigned long
    _page_hashfn(struct inode * inode,
        unsigned long offset)
{
#define i (((unsigned long) inode)/
        (sizeof(struct inode) &
          ~ (sizeof(struct inode) - 1)))
#define o (offset >> PAGE_SHIFT)
#define s(x) ((x)+((x)>>PAGE_HASH_BITS))

        return s(i+o) & (PAGE_HASH_SIZE-1);

#undef i
#undef o
#undef s
}
```

The hash function key is made up of two arguments: the `inode` and the `offset`. The `inode` argument is a

kernel	table size (buckets)	16 scripts	32 scripts	48 scripts	64 scripts	total elapsed
reference	2048	1864.7 s=3.77	1800.8 s=8.51	1739.9 s=3.61	1644.6 s=29.35	50 min 25 sec
13-bit	8192	1875.8 s=5.59	1834.0 s=3.71	1765.5 s=3.01	1683.3 s=17.39	49 min 43 sec
14-bit	16384	1877.2 s=5.35	1830.8 s=3.81	1770.5 s=3.84	1694.3 s=41.42	49 min 35 sec
15-bit	32768	1875.4 s=10.72	1832.4 s=3.97	1770.3 s=3.97	1691.2 s=20.05	49 min 36 sec
offset	16384	1880.0 s=2.78	1843.7 s=14.65	1774.5 s=4.30	1685.4 s=33.46	49 min 40 sec
mult	16384	1876.4 s=6.45	1836.8 s=6.45	1773.7 s=5.20	1691.7 s=25.32	49 min 29 sec
rbtree	N/A	1874.9 s=6.57	1817.0 s=5.59	1755.3 s=3.01	1670.8 s=17.26	50 min 3 sec

Table 1. Benchmark throughput comparison of different hash functions in the page cache hash table. This table compares the performance of several Linux kernels using differently tuned hash tables in the page cache. Total benchmark elapsed time shows the multiplicative hash function improves performance the most.

memory address of the in-core inode that contains the data mapped into the requested page. The `offset` argument is a memory address of the requested page relative to a virtual address space. The result of the function is an index into the page cache hash table.

This simple fast shift-add hash function is surprisingly effective due to the pre-existing randomness of the inode address and offset arguments. Our tests reveal that bucket size remains acceptable as PAGE_HASH_BITS is varied from 11 to 16.

Normally, the `offset` argument is page-aligned, but when the page cache is doubling as the swap cache, the `offset` argument can contain important index-randomizing information in the lower bits. Stephen Tweedie suggests [LKA] that adding the offset again, unshifted, before computing `s()` would improve bucket size distribution problems caused when hashing swap cache pages. Our tests show that adding the unshifted offset reduced bucket size distribution anomolies, at a slight but measurable across-the-board performance cost.

The following tables show relative throughput results for kernels built with hash table tuning modifications. The “reference” kernel is a plain 2.2.5 kernel with a 4000 entry process table. The “13-bit,” “14-bit,” and “15-bit” kernels are plain 2.2.5 kernel with a 4000 entry process table and a 13, 14, and 15-bit (8192, 16384, and 32768 buckets) page cache hash table. The “offset” kernel is just like the “14-bit” kernel, but whose page cache hash function looks like this:

```
return s(i+o+offset) & (PAGE_HASH_SIZE-1);
```

The “mult” kernel is the “14-bit” kernel with a multiplicative hash function instead of the plain additive one:

```
return ((((unsigned long)inode + offset) *
    2654435761UL) >> \
        (32 - PAGE_HASH_BITS)) &
            (PAGE_HASH_SIZE-1);
```

See the section on multiplicative hashing for more about how we derived this function.

Finally, the “rbtree” kernel was derived from a clean 2.2.5 kernel with a special patch applied, extracted from Andrea Archangeli’s 2.2.5-arca10. This patch implements the page cache with per-inode red-black trees, a form of balanced binary tree [CLR], instead of a hash table.

We ran each workload seven times, and take the results from the middle five runs. The results in Table 1 are averages and standard deviations for the middle five benchmark runs for each workload. The timing result is the total length of all the runs for that kernel, including the two runs out of seven that were ignored in the average calculations. Each set of runs for a given kernel is benchmarked on a freshly rebooted system. These are obtained on our dual Pentium Pro using sixteen, thirty-two, forty-eight, and sixty-four concurrent script workloads to show how performance changes between CPU bound and I/O bound workloads. We also want to push the machine into swap to see how performance changes when the page cache is used as a swap cache.

According to our own kernel program counter profiling results, defining PAGE_HASH_BITS as 13 bits is enough to take `find_page()` out of the top kernel

table size, in buckets	average throughput	average throughput, minus first run	maximum throughput	elapsed time
2048	4282.8 s=29.96	4295.2 s=11.10	4313.0	12 min 57 sec
8192	4387.3 s=23.10	4398.5 s=5.88	4407.5	12 min 40 sec
32768	4405.3 s=5.59	4407.4 s=4.14	4413.8	12 min 49 sec

Table 2. Benchmark throughput comparison of different hash table sizes in the page cache hash table. This table shows benchmark performance of our tweaked kernels on large memory hardware. This test shows how performance changes when the data structure is heavily populated, and the system is not swapping.

CPU users during most heavy VM loads on large-memory machines. However, increasing it further can help reduce the real elapsed time required for an average lookup, thus improving system performance even more. As one might expect, increasing the hash table size had little effect on smaller workloads. To show the effects of increased table size on a high-end machine, we ran 128 script benchmarks on our four-way 512M Dell PowerEdge. The kernels used in this test are otherwise unchanged reference kernels compiled with 4000 process slots. The results are averages of five runs on each kernel.

The gains in inter-run variance are significant for larger memory machines. It is also clear that overall performance improves for tables larger than 8192 buckets, although not to the same degree that it improves for a table size increase of 2048 to 8192 buckets.

The “rbtree” kernel performs better than the “reference” kernel. It also scores very well in inter-run variance. A big advantage of this implementation is that it is more space efficient, especially on small machines, as it doesn’t require contiguous pages for a hash table. We predicted the “offset” kernel to perform better when the system was swapping, but it appears to perform worse than both the “mult” and the “14-bit” kernel on the heaviest workload. Finally, the “mult” kernel appears to have the smoothest overall results, and the shortest overall elapsed time.

Because of the overall goodness of the existing hash function, the biggest gain occurs when the page cache hash table size is increased. This has performance benefits for machines of all memory sizes; as hash table size increases, more pages are hashed into buckets that contain only a single page, decreasing average lookup time.

Increasing the page cache hash table’s bucket count even further continues to improve performance, especially for large memory machines. However, for use on generic hardware, 13 bits accounts for 8 pages worth of hash table, which is probably the practical upper limit for small memory machines.

```
Apr 27 17:17:51 pillbox kernel: Buffer cache total lookups: 296481  (hit rate: 54%)
Apr 27 17:17:51 pillbox kernel:  hash table size is 16384 buckets
Apr 27 17:17:51 pillbox kernel:  hash table contains 37256 objects
Apr 27 17:17:51 pillbox kernel:  largest bucket contains 116 buffers
Apr 27 17:17:51 pillbox kernel:  find_buffer() iterations/lookup: 2155/1000
Apr 27 17:17:51 pillbox kernel:  hash table histogram:
Apr 27 17:17:51 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 17:17:51 pillbox kernel:     0    12047        0       0
Apr 27 17:17:51 pillbox kernel:     1     1037     1037       2
Apr 27 17:17:51 pillbox kernel:     2      381      762       4
Apr 27 17:17:51 pillbox kernel:     3      295      885       7
Apr 27 17:17:51 pillbox kernel:     4      325     1300      10
Apr 27 17:17:51 pillbox kernel:     5      399     1995      16
Apr 27 17:17:51 pillbox kernel:     6      188     1128      19
Apr 27 17:17:51 pillbox kernel:     7      303     2121      24
Apr 27 17:17:51 pillbox kernel:     8      160     1280      28
Apr 27 17:17:51 pillbox kernel:     9      169     1521      32
Apr 27 17:17:51 pillbox kernel:    10      224     2240      38
Apr 27 17:17:51 pillbox kernel:    11       64      704      40
Apr 27 17:17:51 pillbox kernel:    12       49      588      41
Apr 27 17:17:51 pillbox kernel:    13       15      195      42
Apr 27 17:17:51 pillbox kernel:    14        3       42      42
Apr 27 17:17:51 pillbox kernel:    15        4       60      42
Apr 27 17:17:51 pillbox kernel:   >15      721    21398     100
```

Histogram 1. Full buffer cache using the old hash function. This histogram demonstrates how poorly the Linux buffer cache spreads buffers across the buffer cache hash table. Most of the buffers are stored in hash buckets that contain more than 15 other buffers. This slows benchmark throughput markedly.

3.2 Buffer cache

Linux holds dirty data blocks about to be written to disk in its buffer cache. The buffer cache hash table in the plain 2.2.5 kernel comprises 32768 buckets, and uses this hash function from fs/buffer.c:

```
#define _hashfn(dev,block)
(((unsigned)(HASHDEV(dev)^block)) &
bh_hash_mask)
```

This function adds no randomness to either argument, simply xor-ing them together, and truncating the result.

Histogram 1 tells the whole story. Histogram 1 was obtained during several heavy runs of our benchmark suite on the dual Pentium Pro. Each histogram divides its output into several columns. First, the "buckets" column reports the observed number of buckets in the hash table containing "size" objects; there are 1037 buckets observed to contain a single buffer in this example. The "buffers" column reports how many buffers are found in buckets of that size, a product of the size and observed bucket count. The "sum-pct" column is the cumulative percentage of buffers contained in buckets of that size and smaller. In other words, in the Histogram 1, 28% of all buffers in the hash table are stored in buckets containing 8 or fewer buffers, and 42% of all buffers were stored in buckets containing 15 or fewer buffers. The number of empty buckets in the hash table is the value reported in the "buckets" column for size 0.

The average bucket size for 37,000+ buffers stored in a 16384 bucket table should be about 3 (that is, **O(n/m)**, where **n** is the number of objects contained in the hash table, and **m** is the number of hash buckets). The largest bucket contains 116 buffers, almost 2 orders of magnitude more than the expected average, even though the hash table is less than twenty-seven percent utilized (16384 total buckets minus 12047 empty buckets, divided by 16384 total buckets gives us 0.27). At one point during the benchmark, the author observed buckets containing more than 340 buffers.

After the benchmark is over, most of the buffers still reside in large buckets; see Histogram 2. Eighty-five percent of the buffers in this cache are contained in buckets with more than 15 buffers in them, even though there are 16167 empty buckets -- an effective bucket utilization of less than two percent!

Clearly a better hash function is needed for the buffer cache hash table. The following table compares benchmark throughput results from the reference kernel (unmodified 2.2.5 kernel with 4000 process slots, as above) to results obtained after replacing the buffer cache hash function with several different hash functions. Here is our multiplicative hash function:

```
#define _hashfn(dev,block) ((((block) *
    2654435761UL) >> SHIFT) &
        bh_hash_mask)
```

We tested variations of this function (SHIFT value is fixed at 11, or varies depending on the table size). We also tried a shift-add hash function to see if the multiplicative hash was really best. The shift-add function comes from Peter Steiner, and uses a shift and subtract ((block << 7) - block) to effectively multiply

```
Apr 27 17:30:49 pillbox kernel: Buffer cache total lookups: 3548568 (hit rate: 78%)
Apr 27 17:30:49 pillbox kernel:  hash table size is 16384 buckets
Apr 27 17:30:49 pillbox kernel:  hash table contains 2644 objects
Apr 27 17:30:49 pillbox kernel:  largest bucket contains 80 buffers
Apr 27 17:30:49 pillbox kernel:  find_buffer() iterations/lookup: 1379/1000
Apr 27 17:30:49 pillbox kernel:  hash table histogram:
Apr 27 17:30:49 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 17:30:49 pillbox kernel:     0    16167        0        0
Apr 27 17:30:49 pillbox kernel:     1      110      110        4
Apr 27 17:30:49 pillbox kernel:     2       10       20        4
Apr 27 17:30:49 pillbox kernel:     3        3        9        5
Apr 27 17:30:49 pillbox kernel:     4        1        4        5
Apr 27 17:30:49 pillbox kernel:     5        0        0        5
Apr 27 17:30:49 pillbox kernel:     6        3       18        6
Apr 27 17:30:49 pillbox kernel:     7        1        7        6
Apr 27 17:30:49 pillbox kernel:     8        6       48        8
Apr 27 17:30:49 pillbox kernel:     9        2       18        8
Apr 27 17:30:49 pillbox kernel:    10        1       10        9
Apr 27 17:30:49 pillbox kernel:    11        2       22       10
Apr 27 17:30:49 pillbox kernel:    12        3       36       11
Apr 27 17:30:49 pillbox kernel:    13        3       39       12
Apr 27 17:30:49 pillbox kernel:    14        3       42       14
Apr 27 17:30:49 pillbox kernel:    15        1       15       15
Apr 27 17:30:49 pillbox kernel:   >15       68     2246      100
```

Histogram 2. Buffer cache using the old hash function, after benchmark is complete. This histogram shows that, even after the benchmark completes, most buffers in the cache remain in hash buckets containing more than 15 other buffers. Additionally, 3,000+ buffers stored in about 220 buckets, although more than 16,000 empty buckets remain. Over time, buffers tend to congregate in large buckets, and system performance suffers.

kernel	table size	average throughput	avg throughput, minus first run	maximum throughput	elapsed time
reference	32768	4282.8 s=29.96	4295.2 s=11.10	4313.0	12 min 57 sec
mult, shift 16	32768	4369.3 s=19.35	4376.4 s=14.53	4393.2	12 min 45 sec
mult, shift 11	32768	4380.8 s=12.09	4382.8 s=11.21	4394.0	12 min 50 sec
shift-add	32768	4388.9 s=21.90	4397.2 s=11.70	4415.5	12 min 31 sec
mult, shift 11	16384	4350.5 s=99.75	4394.6 s=15.59	4417.2	12 min 41 sec
mult, shift 17	16384	4343.7 s=61.17	4369.9 s=17.39	4390.2	12 min 46 sec
shift-add	16384	4390.2 s=22.55	4399.6 s=8.52	4408.3	12 min 37 sec
mult, shift 18	8192	4328.9 s=16.61	4333.7 s=15.05	4349.6	12 min 41 sec
shift-add	8192	4362.5 s=13.37	4362.8 s=14.90	4382.3	12 min 45 sec

Table 3. Benchmark throughput comparison of different hash functions in the buffer cache hash table. We report the results of benchmarking several new buffer cache hash functions in this table. Using a sophisticated multiplicative hash function appears to boost overall system throughput the most.

by a Mersenne prime (block * 127). Multiplication by a Mersenne prime is easy to calculate, as it reduces to a subtraction and a shift operation.

```
#define _hashfn(dev,block) \
        (((block << 7) - block + (block >> 10)
    + (block >> 18)) &
        bh_hash_mask)
```

This series of tests consists of five runs of 128 concurrent scripts on the four-way Dell PowerEdge system. We report an average result for all five runs, and an average result without the first run. The five-run average and the total elapsed time show how good or bad the first run, which warms the system caches after a reboot, can be. The four-run average indicates the steady-state operation of the buffer cache.

On a Pentium II with 512K of level 2 cache, the shift-add hash shows a higher average throughput than the multplicative variants. On CPUs with less pipelining, the race is somewhat closer, probably because the shift-add function, when performed serially, can sometimes take as long as multiplication. However, the shift-add function also has the lowest variance in this test, and the highest first-run throughput, making it a clear choice for use as the buffer cache hash function.

We also tested with smaller hash table sizes to demonstrate that buffer cache throughput can be maintained using fewer buckets. Our test results bear this out; in fact, often these functions appear to work better with fewer buckets. Reducing the size of the buffer cache hash table saves more than a dozen contiguous pages (in the existing kernel, this hash table already consumes a contiguous 32 pages).

Histogram 3 shows what a preferred bucket size

```
Apr 27 18:14:50 pillbox kernel: Buffer cache total lookups: 287696  (hit rate: 54%)
Apr 27 18:14:50 pillbox kernel:  hash table size is 16384 buckets
Apr 27 18:14:50 pillbox kernel:  hash table contains 37261 objects
Apr 27 18:14:50 pillbox kernel:  largest bucket contains 11 buffers
Apr 27 18:14:50 pillbox kernel:  find_buffer() iterations/lookup: 242/1000
Apr 27 18:14:50 pillbox kernel:  hash table histogram:
Apr 27 18:14:50 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 18:14:50 pillbox kernel:     0     2034        0       0
Apr 27 18:14:50 pillbox kernel:     1     3317     3317       8
Apr 27 18:14:50 pillbox kernel:     2     4034     8068      30
Apr 27 18:14:50 pillbox kernel:     3     3833    11499      61
Apr 27 18:14:50 pillbox kernel:     4     2082     8328      83
Apr 27 18:14:50 pillbox kernel:     5      712     3560      93
Apr 27 18:14:50 pillbox kernel:     6      222     1332      96
Apr 27 18:14:50 pillbox kernel:     7       78      546      98
Apr 27 18:14:50 pillbox kernel:     8       46      368      99
Apr 27 18:14:50 pillbox kernel:     9       19      171      99
Apr 27 18:14:50 pillbox kernel:    10        5       50      99
Apr 27 18:14:50 pillbox kernel:    11        2       22     100
Apr 27 18:14:50 pillbox kernel:    12        0        0     100
Apr 27 18:14:50 pillbox kernel:    13        0        0     100
Apr 27 18:14:50 pillbox kernel:    14        0        0     100
Apr 27 18:14:50 pillbox kernel:    15        0        0     100
Apr 27 18:14:50 pillbox kernel:   >15        0        0     100
```

Histogram 3. Full buffer cache using the mult-11 hash function. This histogram of buffer cache hash bucket sizes shows marked improvement. Most buffers reside in small buckets, thus most buffers in the buffer cache can be found after checking fewer than two or three other buffers in the same bucket.

```
Apr 27 18:27:19 pillbox kernel: Buffer cache total lookups: 3530977  (hit rate: 78%)
Apr 27 18:27:19 pillbox kernel:  hash table size is 16384 buckets
Apr 27 18:27:19 pillbox kernel:  hash table contains 2717 objects
Apr 27 18:27:19 pillbox kernel:  largest bucket contains 6 buffers
Apr 27 18:27:19 pillbox kernel:  find_buffer() iterations/lookup: 215/1000
Apr 27 18:27:19 pillbox kernel:  hash table histogram:
Apr 27 18:27:19 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 18:27:19 pillbox kernel:     0    14302        0        0
Apr 27 18:27:19 pillbox kernel:     1     1555     1555       57
Apr 27 18:27:19 pillbox kernel:     2      442      884       89
Apr 27 18:27:19 pillbox kernel:     3       73      219       97
Apr 27 18:27:19 pillbox kernel:     4        5       20       98
Apr 27 18:27:19 pillbox kernel:     5        3       15       99
Apr 27 18:27:19 pillbox kernel:     6        4       24      100
Apr 27 18:27:19 pillbox kernel:     7        0        0      100
Apr 27 18:27:19 pillbox kernel:     8        0        0      100
Apr 27 18:27:19 pillbox kernel:     9        0        0      100
Apr 27 18:27:19 pillbox kernel:    10        0        0      100
Apr 27 18:27:19 pillbox kernel:    11        0        0      100
Apr 27 18:27:19 pillbox kernel:    12        0        0      100
Apr 27 18:27:19 pillbox kernel:    13        0        0      100
Apr 27 18:27:19 pillbox kernel:    14        0        0      100
Apr 27 18:27:19 pillbox kernel:    15        0        0      100
Apr 27 18:27:19 pillbox kernel:   >15        0        0      100
```

Histogram 4. Buffer cache using the mult-11 hash function, after the benchmark is complete. The reader can compare this histogram with the earlier one that reports the buffer cache bucket size distribution after the benchmark has completed. As buffers are removed from the buffer cache, the bucket size distribution remains good when using the multiplicative hash function.

distribution histogram looks like. These runs were made with the mult-11 hash function and a 16384 bucket hash table. This histogram snapshot was made at approximately the same points during the benchmark as the examples above.

After the benchmark completes, the hash table returns to a nominal state. We can also see that the measured iterations per loop average is an order of magnitude less than with the original hash function.

We’d like to underscore some of the good statistical properties demonstrated in Histogram 3. First, the bucket size distributions shown in this histogram approach the shape of a bell curve, suggesting that the hash function is doing a good job of randomizing the keys. The height of the distribution occurs for buckets of size 3 (our expected average), which is about **n/m**, where **n** is the number of stored objects, and **m** is the number of buckets. A perfect distribution centers on the expected average, and has very short tails on either side, only one or two buckets. While the distribution in Histogram 3 is somewhat skewed, observations of tables that are even more full show that the curve becomes less skewed as it fills; that is, as the expected average grows away from zero, the shape of the size distribution more closely approximates the normal distribution. In all cases we’ve observed, the tail of the skew is fairly short, and there appear to be few degenerations of the hash (where one or more very large buckets appear).

Second, in both Histogram 3 and 4, about 68% of all buffers contained in the hash table are stored in buckets containing the expected average number of buffers or less. Sixty-eight percent of all samples is the expected standard deviation. And third, the number of empty buckets in the first example above is only 12.4%, meaning more than 87% of all buckets in the table are used.

3.3 Dentry cache

The Linux 2.2 kernel has a directory entry cache, or dentry cache, that is designed to speed up file system performance by mapping file names directly to the in-core address of the inode struct associated with the file. The plain 2.2.5 kernel uses a hash table with 1024 buckets to manage the dentry cache. A simple shift-add hash function is employed:

```
#define D_HASHBITS     10
#define D_HASHSIZE     (1UL << D_HASHBITS)
#define D_HASHMASK     (D_HASHSIZE-1)

static inline struct list_head * d_hash(
    struct dentry * parent,
        unsigned long hash)
{
    hash += (unsigned long) parent;
    hash = hash ^
        (hash >> D_HASHBITS) ^
        (hash >> D_HASHBITS*2);
    return dentry_hashtable +
        (hash & D_HASHMASK);
}
```

kernel	average throughput	maximum throughput	elapsed time
reference	4282.8 s=29.96	4313.0	12 min 57 sec
14 bit	4375.2 s=25.92	4397.4	12 min 42 sec
mult, shift 11	4368.7 s=62.65	4406.2	12 min 39 sec
mult, shift 17	4375.9 s=10.40	4389.0	12 min 40 sec
shrink	4368.7 s=33.36	4390.7	12 min 40 sec
shrink + mult 11	4380.4 s=13.53	4396.5	12 min 35 sec
shrink + mult 17	4368.5 s=16.21	4383.6	12 min 42 sec

Table 5. Benchmark throughput comparison of different hash functions in the dcache cache hash table. This table shows that increasing the hash table size in the `dentry` cache has significant benefits for system throughput, decreasing benchmark elapsed time by 15 seconds. Other changes decrease elapsed time by only a few seconds.

kernel	average throughput	elapsed time
reference	4282.8 s=29.96	12 min 57 sec
12 bit	4361.3 s= 11.15	12 min 36 sec
mult	4346.0 s=20.87	12 min 52 sec
14 bit	4368.3 s= 20.41	12 min 54 sec

Table 4. Benchmark throughput comparison of different hash functions in the inode cache hash table. Increasing the size of the inode cache hash table has clear performance benefits, as this table shows. Replacing the hash function in this cache actually hurts performance.

The arguments for this function are the address of the parent directory's dentry structure, and a hash value obtained by a simplified CRC algorithm on the target entry's name. This function appears to work fairly well, but we want to improve it nonetheless.

Andrea Arcangeli suggests that shrinking the dcache slightly more aggressively might reduce the number of objects in the table enough to help improve dcache hash lookup times [LKA]. We test this idea by adding a couple of lines from his 2.2.5-arca10 patch: In fs/dcache.c, function `shrink_dcache_memory()`, we replace `prune_dcache(found)` with:

```
prune_dcache(dentry_stat.nr_unused /
    (priority+1));
```

and move the `shrink_dcache_memory()` call in `do_try_to_free_pages()` close to the top of the loop so that it will be invoked more often. In Table 4, we show results from several different kernels. First, results from the reference 2.2.5 kernel are repeated, then a kernel that is like the reference kernel, except the dcache hash table is increased to 16384 buckets, and the xor operations are replaced with addition when computing the hash function. The "shrink" kernel is a 2.2.5 kernel like the "14-bit" kernel except that it more aggressively shrinks the dcache, as explained above. The "mult" kernels use a multiplicative hash function similar to the buffer cache hash function, instead of the existing dcache hash function:

```
static inline struct list_head * d_hash(
    struct dentry * parent,
    unsigned long hash)
{
    hash += (unsigned long) parent;
    hash = (hash * 2654435761UL) >> SHIFT;
    return dentry_hashtable +
        (hash & D_HASHMASK);
}
```

where SHIFT is either 11 or 17. The "shrink+mult" kernels combine the effects of both multiplicative hashing and shrinking the dcache.

The following results are average results from five benchmark runs of 128 concurrent scripts on the four-way Dell PowerEdge. The timing results are the elapsed time for all five runs on each kernel.

Some may argue that shrinking the dcache unnecessarily might lower the overall effectiveness of the cache, but we believe that shrinking the cache more aggressively will help, rather than hurt, overall system performance because a smaller cache allows faster lookups and causes less CPU cache pollution. In combination with an appropriate multiplicative hash function, such as the one used in the "shrink+mult 11" kernel, elapsed time and average throughput stays high enough to make it the fastest kernel benchmarked in this series.

3.4 Inode cache

The dentry cache, described above, provides a fast way of mapping directory entries to inodes. Kernel developers expected the dentry cache to reduce the need for an efficient inode cache. Thus, when the dentry cache was implemented, the inode cache hash table was reduced to 256 buckets (8 bit hash). As we shall see, this has had a more profound impact on system performance than expected.

The inode cache hash function is a shift-add function similar to the dentry cache hash function.

```
#define HASH_BITS       8
#define HASH_SIZE       (1UL << HASH_BITS)
#define HASH_MASK       (HASH_SIZE-1)

static inline unsigned long hash(
    struct super_block *sb,
    unsigned long i_ino)
{
    unsigned long tmp = i_ino |
        (unsigned long) sb;
    tmp = tmp + (tmp >> HASH_BITS) +
        (tmp >> HASH_BITS*2);
    return tmp & HASH_MASK;
}
```

Histogram 5 shows why this table is too small. The hash chains are extremely long. In addition, the hit rate shows that most lookups are unsuccessful, meaning that almost every lookup request has to traverse the entire bucket. The number of iterations per lookup is almost 40!

Even though there are an order of magnitude fewer lookups in the inode cache than there are in the other caches, this cache is still clearly a performance bottleneck. To demonstrate this, we ran tests on four different hash functions. Our reference kernel results (from Table 1) reappear in Table 5 for convenience. The "12-bit" kernel is the same as the reference kernel except that the hash table size has been increased to 4096 buckets. The "mult" kernel has 4096 inode cache hash table buckets as well, and uses the multiplicative hash function introduced above. The "14-bit" kernel is the same as the reference kernel except that the hash table size has been increased to 16384 buckets.

The 12 bit hash table is the clear winner. Increasing the hash table size further helps performance slightly, but also increases inter-run variance to such an extent that total elapsed time is longer than for the "12-bit" kernel. Adding multiplicative hashing doesn't help much here because the table is already fairly full and well-balanced.

4. Combination testing

In this section, we optimize all four hash tables, and benchmark the resulting kernels. Our benchmarks are ten 128 script runs on the four-way Dell.

We selected optimizations among the best results shown above, then tried them in combination. We find that there are some performance relationships among the various caches, so we show the results for the best combinations that we tried.

```
Apr 27 17:23:31 pillbox kernel: Inode cache total lookups: 189321  (hit rate: 3%)
Apr 27 17:23:31 pillbox kernel:  hash table size is 256 buckets
Apr 27 17:23:31 pillbox kernel:  hash table contains 9785 objects
Apr 27 17:23:31 pillbox kernel:  largest bucket contains 54 inodes
Apr 27 17:23:31 pillbox kernel:  find_inode() iterations/lookup: 38978/1000
Apr 27 17:23:31 pillbox kernel:  hash table histogram:
Apr 27 17:23:31 pillbox kernel:   size  buckets    inodes sum-pct
Apr 27 17:23:31 pillbox kernel:     0        0        0       0
Apr 27 17:23:31 pillbox kernel:     1        0        0       0
Apr 27 17:23:31 pillbox kernel:     2        0        0       0
Apr 27 17:23:31 pillbox kernel:     3        0        0       0
Apr 27 17:23:31 pillbox kernel:     4        0        0       0
Apr 27 17:23:31 pillbox kernel:     5        0        0       0
Apr 27 17:23:31 pillbox kernel:     6        0        0       0
Apr 27 17:23:31 pillbox kernel:     7        0        0       0
Apr 27 17:23:31 pillbox kernel:     8        0        0       0
Apr 27 17:23:31 pillbox kernel:     9        0        0       0
Apr 27 17:23:31 pillbox kernel:    10        0        0       0
Apr 27 17:23:31 pillbox kernel:    11        0        0       0
Apr 27 17:23:31 pillbox kernel:    12        0        0       0
Apr 27 17:23:31 pillbox kernel:    13        0        0       0
Apr 27 17:23:31 pillbox kernel:    14        0        0       0
Apr 27 17:23:31 pillbox kernel:    15        0        0       0
Apr 27 17:23:31 pillbox kernel:   >15      256     9785     100
```

Histogram 5. Full inode cache using the old hash function. This histogram shows what happens when too many objecst are stored in an undersized hash table. Every inode in this hash table resides in a bucket that contains, on average, 37 other objects. Combined with the very low hit rate, this results in a significant negative performance impact.

“Reference” kernel

The “Reference” kernel is a stock 2.2.5 Linux kernel with 4000 process slots:

- a 32768 bucket buffer hash table with a one-to-one hash function
- a 2048 bucket page hash table with a simple shift-add hash function
- a 256 bucket inode hash table with a simple shift-add hash function
- a 1024 bucket dentry hash table with a simple shift-add hash function

Kernel “A”

Kernel “A” is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket hash table using the multiply and shift-by-11 hash function
- a 8192 bucket page cache with the multiplicative hash function described in the page cache section
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function.

Kernel “B”

Kernel “B” is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket buffer hash table with Peter Steiner’s shift-add hash function
- a 8192 bucket page cache with the multiplicative hash function described in the page cache section
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function.

Kernel “C”

Kernel “C” is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket hash table using the multiply and shift-by-11 hash function
- a 8192 bucket page cache with the reference kernel’s hash function
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function.

Kernel “D”

Kernel “D” is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket has table using the multiply and shift-by-11 hash function
- a 8192 bucket page cache with the offset hash function described above
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function

Examining Table 6, we’d like to select a combination that reduces inter-run variance and elapsed time, as well as maximizes throughput and minimizes hash table memory footprint. While kernel “C” offers the highest maximum throughput, its inter-run variance is also largest. On the other hand, kernel “D” has the second highest average throughput, the shortest elapsed time, and the best inter-run variance. This seems like a reasonable compromise.

A patch against Linux 2.2.5 is available for kernel “D” at our website. See Appendix A.

5. Multiplicative hashing

Hash function alternatives include:

- Using an untransformed key
- Modulus hashing
- Multiplicative hashing
- Using an inexpensive but sub-optimal shift-

kernel	average throughput	maximum throughput	elapsed time
Reference	4300.7 s=15.73	4321.1	26 min 41 sec
Kernel A	4582.9 s=12.55	4592.8	25 min 24 sec
Kernel B	4577.9 s=16.22	4602.0	25 min 18 sec
Kernel C	4596.2 s=22.30	4619.5	25 min 18 sec
Kernel D	4591.3 s=10.98	4608.9	25 min 15 sec

Table 6. Benchmark throughput comparison of multiple kernel hash optimizations. Combining improvements in each of the four caches we studied results in an elapsed time improvement of almost a minute and a half.

add hash function

- Using a “correct” shift-add hash function
- Using a hash function driven by one or more random tables
- Architecture-specific hash functions (*e.g.* multiplication on fast, modern processors, and something else on older processors)

Multiplicative hashing is a form of modulus hashing that is less expensive because the results are often as good but a multiplication operation is used instead of a division operation. Multiplicative hashing is controversial because of the expense of multiplication instructions on some hardware types. For example, on 68030 CPUs, popular in old Sun and Macintosh computers, multiplication requires up to 44 CPU cycles for a 32-bit multiplication, whereas a memory load only requires an extra 2 cycles per instruction [Mot]. On a hardware architecture like the 68030 that has little caching, fast load times compared to CPU operations, and expensive multiplication, a multiplicative hash might be inferior even if it cuts the average number of loop iterations per lookup request by a factor of four or more.

However, it turns out that several of the alternatives are just as expensive, or even more expensive, than multiplicative hashing. Random table-driven hash functions require several table lookups, and several shifts, logical AND operations, and additions. An e-mail message from the linux-kernel mailing list [LKA] explains the problem; see Appendix B.

On our example 68030, shifting requires between 4 and 10 cycles, and addition operations aren’t free either. If the instructions that implement the hash function are many, they will likely cause instruction cache contention that will be worse for performance than a multiplication operation. In general, a proper shift-add hash function is almost as expensive in CPU cycles as a multiplicative hash. On a modern superscalar processor, shifting and addition operations can occur in parallel as long as there are no address generation interlocks (AGIs). An AGI occurs when the results of one operation are required to form an address in a later operation that might otherwise have been parallelized by superscalar CPU hardware [Schmit, P2]. AGIs are much more likely for a table-driven hash function.

Multiplicative hash functions are often very concise. The hash functions we tried above, for example, compile to three instructions on ia32, comprising 15 bytes. Included in the 15 bytes are all the constants involved in the calculation, leaving only the key itself to be loaded as data. In other words, the whole hash function fits into a single line in the CPU’s instruction cache on contemporary hardware. The shift-add hash functions are generally lengthy, requiring several cache lines to contain, multiple loads of the key, and register allocation contention.

The question becomes, finally, how many CPU cycles should be spent by the hash function to get a reasonable bucket size distribution? In most practical situations, a simple shift-add function suffices. However, one should always test with actual data before deciding on a hash function implementation. Hashing on block numbers, as the Linux buffer cache does, turns out to require a particularly good hash function, as disk block numbers exhibit a great deal of regularity.

5.1 A Little Theory

Our multiplicative hash functions were derived from [Knuth], p. 513ff. The theory posits that machine multiplication by a large number that is likely to cause overflow is the same as finding the modulus by a different number. We won’t repeat Knuth here, but suffice it to say that choosing such a number is complicated. In brief, our choice is based on finding a prime that is in golden ratio to the machine’s word size (2 to the 32nd in our case). Primality isn’t strictly necessary, but it adds certain desirable qualities to the hash function. See Knuth for a discussion of these desirable qualities.

We selected 2654435761 as our multiplier. It is prime, and its value divided by 2 to the 32nd is a very good approximation of the golden ratio [CRC, LKP].

(sqrt(5) - 1) / 2 = 0.618033989

2654435761 / 4294967296 = 0.618033987

To obtain the best effects of this “division” we need to choose the correct shift value. This is usually the word size, in bits, minus the hash table size, in bits. This shifts the most significant bits of the result of the “division” down to where they can be used as the hash table index, preserving the greatest effects of the golden ratio. However, sometimes experimentation reveals a better shift value for a given set of input data.

6. Conclusions and Future Work

Careful selection and optimization of kernel hash tables can boost performance significantly, and improve inter-run variance as well, maximizing system throughput. However, selecting a good hash function and benchmarking its effectiveness can be tedious. Usually, the most significant performance optimization comes from increasing the size of a hash table.

To extend this study the cache instrumentation patch should be re-written to use a file in /proc instead of writing to system console log, and should be integrated into the stock kernel as a "Kernel Hacking" configuration option. The tuning patch should be benchmarked on 64-bit hardware to see if another constant needs to be chosen there. A benchmark run on older architectures, such as MC68000, should determine if these changes will seriously degrade performance on older machines.

We could also investigate the performance difference between inlining the page cache management routines (which eliminates the subroutine call overhead) and leaving them as stand-alone routines (which means they have a smaller L1 cache footprint). A separate swap cache hash function might also optimize the separate uses of the page cache hash tables.

As well, there are still open questions about why shrinking the dentry cache more aggressively can help performance. A study could focus on the cost of a dentry cache miss versus the cost of a page fault or buffer cache miss. Discovering alternative ways of triggering a dentry cache prune operation, or alternate ways of calculating the prune priority, may also be prudent.

Finally, there is still opportunity to analyze even more carefully the real keys and hash functions in use in several of the tables we've analyzed here, as well as several tables we didn't visit in this report, such as the uid and pid hash tables, and the vma data structures.

7. Acknowlegements

The author gratefully acknowleges the input and contributions of the following persons: Peter Steiner, Andrea Arcangeli, Iain McClatchie , Paul F. Dietz , Janos Farkas, Dr. Horst von Brand, and Stephen C. Tweedie, as well as the many others who contributed directly and indirectly to the work described in this report. Special thanks go to Dr. Charles Antonelli and Prof. Gary Tyson for providing the hardware benchmarked in this report. Thanks also to the reviewers for their input.

References

[LKA] linux-kernel archives

[CRC] *CRC Standard Mathematical Tables*, 25th Edition, William H. Beyer, Ed., CRC Press, Inc., 1978.

[CLR] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[Hen] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann, 1996.

[Knuth] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison-Wesley, 1998.

[Schmit] M. L. Schmit, *Pentium(tm) Processor Optimization Tools*, Academic Press, Inc., 1995.

[SPEC] Standard Performance Evaluation Corporation, System Development Multitask Benchmark SPEC, 1991

[Mot] *MC68030 User's Manual, Volume 2*, Motorola, Incorporated, 1998.

[P2] Pentium II processor reference manuals.

[LKP] *The Largest Known Primes*, www.utm.edu/research/primes/largest.html, 1998.

Appendix A: Kernel patches

Our research has identified some kernel modifications that may be of use to others in the Linux kernel development community. For more information on these modifications, see the Linux Scalability Project web site:

`http://www.citi.umich.edu/projects/linux-scalability`

Appendix B: E-mail

Date: Thu, 15 Apr 1999 15:01:54 -0700
From: Iain McClatchie
To: Paul F. Dietz
Cc: linux-kernel@vger.rutgers.edu
Subject: Re: more on hash functions

I got a few suggestions about how to use multiple lookups with a single table. All the suggestions make the hash function itself slower, and attempt to fix an issue -- hash distribution -- that doesn't appear to be a problem. I thought I should explain why the table lookup function is slow.

A multiplication has a scheduling latency of either 5 or 9 cycles on a P6. Four memory accesses take four cycles on that same P6. So the core operations for the two hash function are actually very similar in delay, and the table lookup appears to have a slight edge. The difference is in the overhead.

A multiplicative hash, at minimum, requires the loading of a constant, a multiplication, and a shift. Egcs actually transforms some constant multiplications into a sequence of shifts and adds which may have shorter latency, but essentially, the shift (and nothing else) goes in series with the multiplication and as a result the hash function has very little latency overhead.

A table lookup hash spends quite a lot of time unpacking the bytes from the key, and furthurmore uses a load slot to unpack each byte. This makes for 8 load slots, which take 1 cycle each. Even if fully parallelized with unpacking, we end up with a fair bit of latency. Worse yet, egcs runs out of registers and ends up shifting the key value in place on the stack twice, which gobbles two load and two store slots.

Bottom line: CPUs really suck at bit-shuffling and even byte-shuffling. If there is some clever way to code the byte unpacking in the table lookup hash function, perhaps using the x86's trick register file, it might end up faster than the multiplicative hash.

-Iain